Oops! What about a Million Kernel Oopses?

Lisong Guo, Peter Senna Tschudin, Kenji Kono, Gilles Muller, Julia Lawall

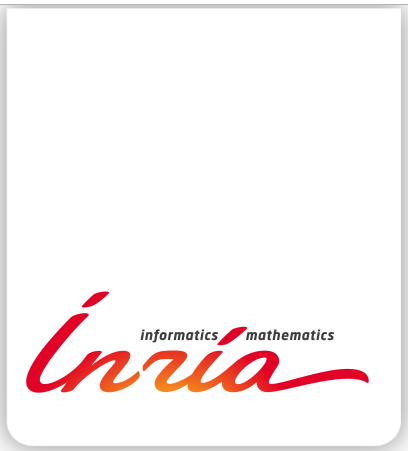
RESEARCH REPORT

N° 436

June 2013

Project-Teams REGAL and LIP6

ISSN 0249-6399 ISRN INRIA/RR--436--FR+ENG



Oops! What about a Million Kernel Oopses?

Lisong Guo*, Peter Senna Tschudin*, Kenji Kono†,
Gilles Muller*, Julia Lawall*

Project-Teams REGAL and LIP6

Research Report n° 436 — June 2013 — 24 pages

Abstract: When a failure occurs in the Linux kernel, the kernel emits an "oops", summarizing the execution context of the failure. Kernel oopses describe real Linux errors, and thus can help prioritize debugging efforts and motivate the design of tools to improve the reliability of Linux code. Nevertheless, the information is only meaningful if it is representative and can be interpreted correctly.

In this paper, we study a repository of kernel oopses collected over 8 months by Red Hat. We consider the overall features of the data, the degree to which the data reflects other information about Linux, and the interpretation of features that may be relevant to reliability. We find that the data correlates well with other information about Linux, but that it suffers from duplicate and missing information. We furthermore identify some potential pitfalls in studying features such as the sources of common faults and common failing applications.

Key-words: Linux, kernel oops, debugging

* lisong.guo@lip6.fr, peter.senna@lip6.fr, gilles.muller@lip6.fr, julia.lawall@lip6.fr
† kono@ics.keio.ac.jp

**RESEARCH CENTRE
PARIS – ROCQUENCOURT**

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Oops ! Que faire d'un Million de Kernel Oopses ?

Résumé : Lorsqu'une défaillance survient dans le noyau de Linux, le noyau émet un rapport de «oops» qui résume le contexte d'exécution à ce moment. Les «kernel oopses» décrivent des vrais problèmes de Linux. Ils peut aider à fixer des priorités pour les tâches de mise au point et motiver le développement d'outils permettant d'améliorer la fiabilité du code de Linux. Néanmoins, les informations des oops ne sont significatives que si elles sont représentatifs et sont interprétées correctement.

Dans cet article, nous étudions les kernel oopses mémorisés pendant huit mois par Red Hat. Nous considérons les caractéristiques générales de ces rapports, la corrélation entre les rapports et d'autres informations éxtérieures, et l'interprétation des caractéristiques liée à la fiabilité de Linux. Notre étude montre principalement (i) que les kernel oopses sont corrélés aux informations extérieures, (ii) les kernel oopses souffrent de problémes de duplication et d'absence de rapports. Par ailleurs, nous avons identifiés des pièges á éviter dans l'étude des sources des fautes usuelles et dans les applications fréquement défaillantes.

Mots-clés : Linux, kernel oops, debugging

1 Introduction

Crash report repositories, as are maintained for software such as Windows [1] and Mac OS X [2], collect crash reports from users as they occur. The data in such repositories thus has the potential to help developers and software researchers alike identify the important trouble spots in the software implementation. As a result, such data has been used for tasks such as prioritizing debugging efforts, detecting the emergence of malware, and monitoring the resurgence of bugs [3].

We consider crash reports in the context of the Linux kernel. Linux is an open-source operating system that is used in settings ranging from supercomputers to embedded systems. The critical nature of many of these usage contexts means that it is important to understand and quickly address the errors in the Linux kernel that are encountered by real users. Between 2007 and 2010, a repository collecting Linux crash reports, commonly known as *oops messages*, or just *oopses*, was maintained under the supervision of `kernel.org`, the main website for distributing the Linux kernel source code. However, due to hardware limitations, that data was difficult to access, and we are not aware of any systematic study of Linux kernel oopses that was made at that time. That data furthermore now appears to be lost, due to unavailability of the underlying database.

Since September 2012, Red Hat has revived the collection of Linux kernel oopses. This repository reopens the possibility of using kernel crash reports to understand the real errors encountered by Linux kernel users. Nevertheless, drawing conclusions from a set of crash reports requires understanding the semantics of the reports themselves and the degree to which the reports are representative of real errors. The Linux kernel has a number of properties that make interpreting oops messages challenging and that potentially call into question the repository data's representativeness.

The first challenge in interpreting the Linux oops reports is that of *code diversity*. The Linux kernel is highly configurable, exists in many versions, and is packaged into many distributions. Quite different code may be executed by users relying on different hardware architectures, different filesystems, different devices, etc. Furthermore, different Linux distributions, such as Debian and Fedora, have different strategies for choosing a kernel: Debian uses older stable versions, while Fedora keeps up with the most recent releases. Thus, a user having one kind of hardware and running one Linux distribution may encounter a quite different set of errors than a user with another.

The second challenge in interpreting the Linux oops reports is the issue of *oops transmission*. For security and performance reasons, the Linux kernel does not itself submit oopses to the repository, but only causes them to be written to a kernel log file. Different Linux distributions provide different user-level applications for retrieving oopses from these log files and submitting them to the repository. Thus, the set of reports that appears in the repository depends on the strategies taken by these tools, which may include asking user permission.

The third challenge in interpreting the Linux oops reports is the issue of *optimization*. As performance-critical, low-level code, the Linux kernel incorporates optimizations and execution strategies that are not typically found in application code. These optimizations and execution strategies affect the reliability of the oops data and complicate its interpretation.

In this paper, we study how we can interpret Linux kernel oopses to draw valid conclusions about Linux kernel reliability. To this end, we perform a study of over 187,000 Linux kernel oopses, collected in the Red Hat repository between September 2012 and April 2013. We first study properties of the data itself, then correlate properties of the data with independently known information about the state of the Linux kernel, and then consider some features of the data that can be relevant to understanding the kinds of errors encountered by real users and how these features can be accurately interpreted. The main lessons learned are as follows:

- The number of oopses available for different versions varies widely, depending on which Linux distributions have adopted the version and the strategies used by the associated oops submission tools.

- The repository may furthermore suffer from duplicate and missing oopses, although the available information does not permit identifying either accurately.

- Identifying the service causing a kernel crash may require considering the call stack, to avoid merging oopses triggered in generic utility functions. The call stack, however, may contain stale information, due to kernel compilation options.

- Analyses of the complete stack must take into account that the kernel maintains multiple stacks, only one of which reflects the current process's or interrupt's execution history.

- Kernel executions may become tainted, indicating the presence of a suspicious action in the kernel's execution history. Oopses from tainted kernels may not reflect independent problems in the kernel code.

The rest of the paper is organized as follows. Section 2 illustrates the key features of a kernel oops, and the workflow around the generation and transmission of a kernel oops. Section 3 gives an overview of the data and studies its internal consistency, focusing on the possibility of duplicate or missing oopses. Section 4 compares properties of the oopses to existing information about the Linux kernel, including versions and their release date, distributions and their associated versions, hardware architectures, and the most error-prone services. Then, Section 5 studies some features of the data that can be relevant to understanding kernel reliability, including the frequency of different error types, failing functions, reasons for entering the kernel, and the impact of previous kernel events. Section 6 then describes the threats to the validity of our study. Finally, we consider related work in Section 7 and conclude in Section 8.

2 Background

We first present the key features of kernel oopses, the code that triggers their generation, and the workflow that results in their being submitted to the Red Hat repository.

2.1 Key features of a kernel oops

A kernel oops documents the internal state of the Linux kernel at the time of a failure. It is represented in plain text, and comprises a number of fields. Each field represents one aspect of the system state in the form of a key-value pair.

Fig. 1 shows a (slightly simplified) sample kernel oops[1] from the Red Hat kerneloops repository. This example illustrates a typical kernel oops generated by x86 code in the case of a runtime exception. We describe its key features, highlighted in boldface.

Oops description A kernel oops begins with a description of the cause of the oops. Our oops was caused by NULL pointer dereference (lines 1).

[1] ID in the kerneloops repository: 5095a67440bfca031f0007e3

```
1  BUG: unable to handle kernel NULL pointer dereference at (null)
2  IP: [<c10a1ca1>] anon_vma_link+0x24/0x2b *pde = 00000000
3  Oops: 0002 [#3] SMP
4  last sysfs file: /sys/devices/LNXSYSTM: 00/LNXSYBUS:00 /PNP0C0A:
     00/power_supply/BAT1/charge_full
5  Modules linked in: rndis_wlan rndis_host cdc_ether...
6     [last unloaded: scsi_wait_scan]
7  Pid: 2452, comm: gnome-panel Tainted: G D (2.6.32-5-686 #1) Aspire 5920
8  EIP: 0060: [<c10a1ca1>] EFLAGS: 00010246 CPU: 0
9  EIP is at anon_vma_link+0x24/0x2b
10 EAX: f6f84404 EBX: f6f84400 ECX: eb4aa5b4 EDX: 00000000
11 ESI: eb4aa580 EDI: eb4aa5d8 EBP: ef76a5d8 ESP: f61c3eb8
12 DS: 007b ES: 007b FS: 00d8 GS: 00e0 SS: 0068
13 Process gnome-panel (pid: 2452, ti=f61c2000 task=ef4a1100 task.ti=f61c2000)
14 Stack:
15   00000006 ef76a630 c102efe8 d42a7a40 00000000 00000004...
16 Call Trace:
17   [<c102efe8>] ? dup_mm+0x1d5/0x389
18   [<c102fb0c>] ? copy_process+0x91b/0xf2d
19   [<c1030258>] ? do_fork+0x13a/0x2bc
20   [<c10b1f41>] ? fd_install+0x1e/0x3c
21   [<c10b9504>] ? do_pipe_flags+0x8a/0xc8
22   [<c113c603>] ? copy_to_user+0x29/0xf8
23   [<c1001dae>] ? sys_clone+0x21/0x27
24   [<c10030fb>] ? sysenter_do_call+0x12/0x28
25 Code: 02 31 db 89 d8 5b c3 56 89 c6 53 8b 58 3c 85 db...
26 EIP: [<c10a1ca1>] anon_vma_link+0x24/0x2b SS: ESP 0068: f61c3eb8
27 CR2: 0000000000000000
28 ---[ end trace 4dbb248fc567ac92 ]---
```

Figure 1: A sample kernel oops

Table 1: The information found in common kinds of oopses

Oops type	id	Version	Taint	Die cnt	Cmd	Stack top	Call trace	general cause
CNTXT_BLOCK		×	×		×		×	*blocking call in a non-blocking context*
CNTXT_SCHED		×	×		×		×	*call schedule in an atomic context*
CNTXT_CALL		×	×		×		×	*func call violates context dependent rules*
PT_MAP		×	×		×		×	*invalid entry in the page mapping table*
PT_STATE		×	×		×		×	*inconsistent or invalid page table*
BUG_ON	×	×	×	×	×	×	×	*assertion failure*
INV_PTR	×	×	×	×	×	×	×	*null pointer/bad page dereference*
WARN	×	×	×		×	×	×	*warning*
SOFT_LOCK		×	×		×	×	×	*one CPU is stuck*
GPF	×	×	×	×	×	×	×	*violation of hardware protection mechanisms*

Error site The *IP* (Instruction Pointer) field indicates the name of the function being executed at the time of the oops, the binary code offset at which the oops occurred in that function, and the binary code size of that function. This information is also indicated by the value of the registers

EIP for the 32-bit x86 architecture and RIP for the 64-bit x86 architecture. Lines 2, 9 and 26 each show that in our example oops, the error occurred within the function `anon_vma_link`.

Die counter The *Oops* field includes information about the die counter, between square brackets, which indicates how many serious errors have occurred since the system was booted. Our oops is the third such error (line 3).

Process name The *comm* field indicates the name of the process being executed when the error occurred. Our oops was generated during the execution of `gnome-panel` (line 13).

Taint The *Tainted* field gives a sequence of characters, amounting to a bitmap, recording whether some events have previously occurred in the kernel execution. In our example, the Tainted field on line 7 contains `'G'` indicating that no proprietary module has been loaded into the kernel and `'D'` indicating that a kernel oops has previously occurred. The latter is consistent with the value of the die counter.

Version Following the taint bits, the *Tainted* field indicates the build version of the Linux kernel and possibly the machine model, defined by the vendor. Line 7 shows that our oops was generated by Linux version 2.6.32.

Call trace The *call trace* contains the list of return pointers into the sequence of nested function calls that led to the oops. Lines 16-24 show the call trace of our oops.

Oops ID A kernel oops may end with a string of the form `--[ end trace XXXX ]--`, where `XXXX` represents the identifier associated with the kernel oops. The identifier (*oops_ id*) is represented as a 16-character hexadecimal string. It represents the current value of a global variable that is initialized to a 64-bit random number on the first oops within a given boot and is then incremented each time an oops containing an oops id is generated. Line 28 shows the oops id of our oops.

2.2 Types of Kernel Oopses

While many of the kernel oopses in the Red Hat repository contain the above features, this is not always the case. Indeed, a kernel oops is generated by ad-hoc print statements in the kernel code. Thus, the format of an oops can vary for each error type. Furthermore, because oopses include architecture-specific information, such as register names and memory addresses, the format of a kernel oops can vary by architecture.

Most of the text in the oops shown in Table 1 is generated by the function `__die`, shown in Fig. 2. This function is used for most severe errors. In `__die`, most of the work is done by the call to `show_registers` (line 7), which prints the registers but also prints other information, including the the call trace. The function `__die`, however, is not used for all kinds of errors. For example, when the memory manager detects a page containing invalid flags, it uses `bad_page` (Fig. 3) to generate an oops. `bad_page` does not use `show_registers`. Instead it uses `dump_stack`, which prints the call trace, but not the register information. Finally, a further variation in the oops structure is introduced by the fact that generation of the call trace is ultimately controlled by a configuration flag. Table 1 lists a collection of common x86 error types and the features that may be included in the corresponding oopses.

```
1  int __kprobes __die(const char *str, struct pt_regs *regs, long err) {
2    unsigned short ss; unsigned long sp;
3    printk(KERN_EMERG "%s: %04lx [#%d] ",str,err&0xffff,++die_counter);
4    printk("SMP \n");
5    if (notify_die(DIE_OOPS, str, regs, err, ...) == NOTIFY_STOP)
6      return 1;
7    show_registers(regs);
8    ... // initialize ss, sp
9    printk(KERN_EMERG "EIP: [<%08lx>] ", regs->ip);
10   print_symbol("%s", regs->ip);
11   printk(" SS:ESP %04x:%08lx\n", ss, sp);
12   return 0;
13 }
```

Figure 2: Function `__die` (simplified), called on many kinds of system faults

```
1  static void bad_page(struct page *page) {
2    ... // code to limit the frequency of reports
3    printk(KERN_ALERT "BUG: Bad page state in process %s pfn:%05lx\n",
4      current->comm, page_to_pfn(page));
5    printk(KERN_ALERT
6      "page:%p flags:%p count:%d mapcount:%d mapping:%p index:%lx\n",
7      page, (void *)page->flags, page_count(page),
8      page_mapcount(page), page->mapping, page->index);
9    dump_stack();
10 out:
11   __ClearPageBuddy(page);
12   add_taint(TAINT_BAD_PAGE);
13 }
```

Figure 3: Function `bad_page` (simplified), called on detecting a bad page

In addition to `__die` and to ad hoc oops generating functions, such as `bad_page`, the Linux kernel also provides the oops-generating macros `BUG` and `WARN`, and variants, for generating bug and warning messages, respectively. These macros are executed for diverse reasons, but generate their oopses in a fixed format. We designate oopses generated by the `WARN` macros as warnings and all others as bugs.

2.3 Workflow around Kernel Oopses

The life cycle of a kernel oops is composed of three phases: *generation*, *collection*, and *submission*. Our understanding of this workflow is based on our study of the related tools and on our experiments in which we generated an oops and observed its path to the repository.

The Linux kernel itself is in charge of *generation*, which is ultimately performed through the kernel printing function `printk`. This function writes to the kernel message buffer (*/proc/kmsg*). The generated message is then asynchronously fetched by the kernel service *'klogd'*, which outputs the contents of the buffer to the system logging files. This strategy implies that some oops messages may be missed or corrupted, if *e.g.*, the message buffer overflows, or if the kernel crashes before the message is picked up by *klogd*.

Collection and *submission* are then taken care of by tool sets that are bundled with various distributions of Linux. Fedora provides *ABRT* (the Automatic Bug Reporting Tool) [4], which collects C/C++ and Python application crash reports, and as well as kernel oopses. *ABRT* consists of several services, including abrtd (daemon to detect crashes), abrt-gui and abrt-applet (GUI for the management of error reports), and abrt-oops (kernel oops parser). Debian and Ubuntu provide *kerneloops* [5], which is dedicated to kernel oopses. *kerneloops* includes the

kerneloops daemon, kerneloops-applet and kerneloops-submit, to collect kernel oopses, to ask for permission to submit them, and to submit them, respectively. Different toolsets have different strategies for storing, parsing, and submitting kernel oopses. For example, *ABRT* truncates the end marker of a kernel oops including the *oops_id*, while *kerneloops* preserves this information.

The kernel oops toolsets submit oopses to both the public `kerneloops.org` [6] repository and the corresponding repository for each Linux distribution. *ABRT* first parses kernel oopses from system logging files, stores them in its local file system, and then asks the user whether to submit them. It keeps the parsed kernel oopses persistent and hashes the content to detect duplicates. *Kerneloops* keeps oopses in memory after parsing, and detects duplicates only within these known oopses. *Kerneloops* thus risks submitting duplicate oopses if it is restarted. We have reproduced this behavior.

3 Properties of the raw data

We now study of the Red Hat kernel oops repository data itself. We first give an overview of the sources of these kernel oopses, in terms of the associated Linux kernel version and the associated Linux distribution. We then consider the possibility of duplicate or missing oopses.

Oops origins As shown in Table 1, most oops kinds contain architecture-specific information, such as register names and memory addresses. We consider only architecture-specific oopses for the x86 processor family, including both 32-bit and 64-bit x86 architectures, as well as all non architecture specific oopses. This amounts to over 99% of the available 187,342 oopses.

Fig. 4 shows the number of reports for each Linux kernel version represented in the repository. The version information is taken from the Tainted field, as described in Section 2.1. The Linux kernel is made available as a series of subreleases, starting with a series of "rc" (release candidate) versions before the primary release, followed by the release itself, followed by a series of bug-fixing releases, having increasing minor release numbers (*e.g.*, 2.6.32.14 or 3.6.4). We do not distinguish between these subreleases. For 7% of the reports, there is no version information. These reports are typically associated with oopses for which the entire report is a single line of text indicating the problem.

Fig. 5 shows the number of reports per Linux distribution, with the Fedora information broken down into the individual Fedora releases. Oopses contain no specific field for distribution information. Instead, we try to deduce the distribution from the version information. Some distributions, such as Fedora or Ubuntu, include a string indicating the distribution explicitly, with Fedora also including the number of the Fedora release. In other cases, such as Debian and Suse, we have used other information, such as filename paths or comments related to the version name found on the Internet. We find that most reports are for Debian or Fedora. This is likely due to the support in these distributions for kernel oops reporting, via the kerneloops and ABRT utilities, rather than the popularity or reliability of the distributions themselves.

The number of reports from Fedora varies widely by release, with Fedora 11, 17 and 18 having the most reports. The number of reports seems to be mainly affected by the strategy taken by Fedora for collecting and transmitting the reports. For example, whether the oops reporting tool is installed by default, and whether it asks the user for permission to send a report or whether reports are sent automatically. Red Hat appears to have experimented with sending more and more reports automatically in Fedora 17 and 18, *i.e.*, once the new repository was available. Nevertheless, this effort seems to have been rolled back at the end of the considered period, due to an outage of the repository between late February and early March 2013.

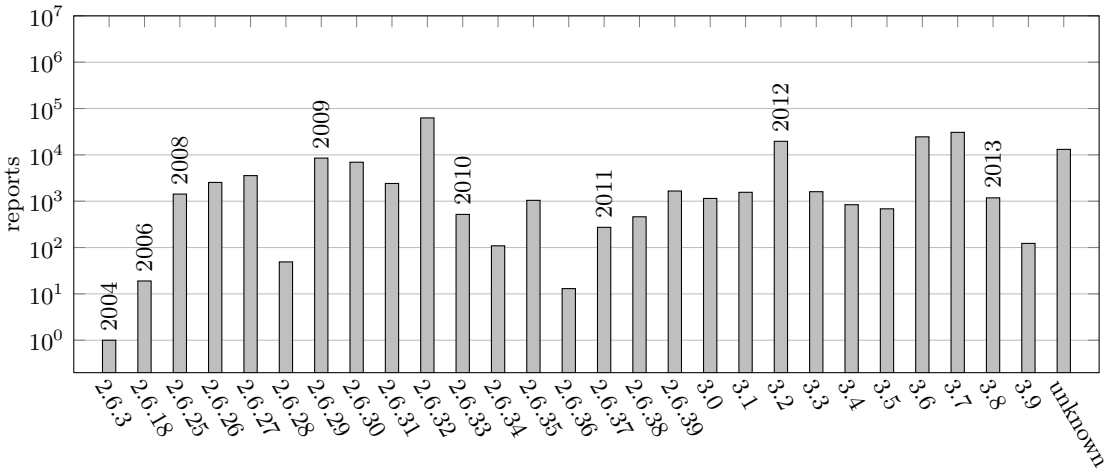


Figure 4: Number of reports per Linux version (log scale) and release year

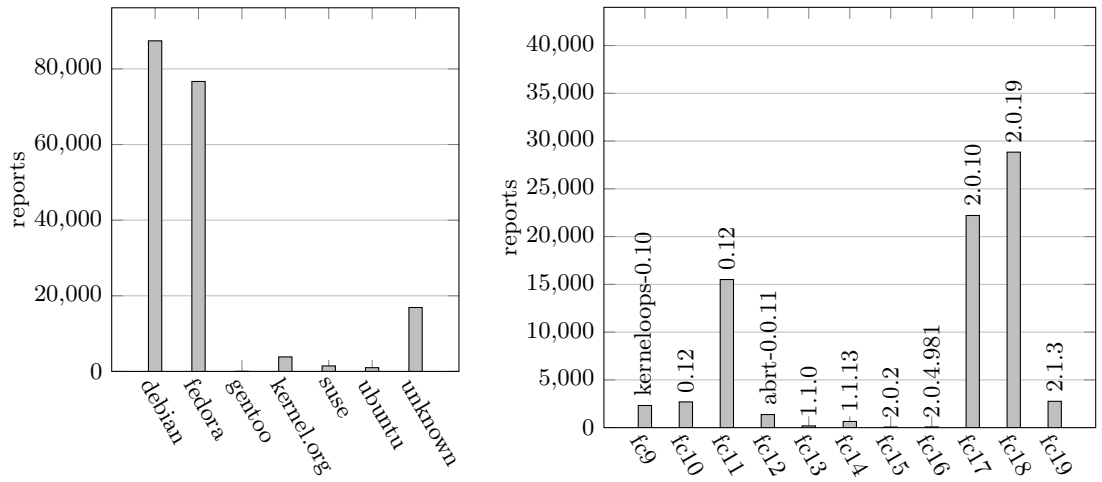


Figure 5: Number of reports per Linux distribution and reporting tool used

In some cases, the version information contains only the number of the Linux version, with no added distribution information. In these cases, we consider that the kernel is not associated with a particular distribution, and instead has been specifically compiled and installed by the user from the code at the Linux kernel website, kernel.org. Such oopses may come from more advanced users.

Duplicate oopses Ideally, each error that occurs in the system would correspond to exactly one oops in the repository. In practice, however, we find many probable duplicates.

Concretely, we have observed the following kinds of probable duplicates: *i.* Identical reports. We detect these by using the SHA1 hash of each original raw message. *ii.* Reports that are identical modulo fragments of kernel time stamps or log level markers. Such text is normally removed by the parsers of the kernel oops tool sets. For duplicate detection, we remove these

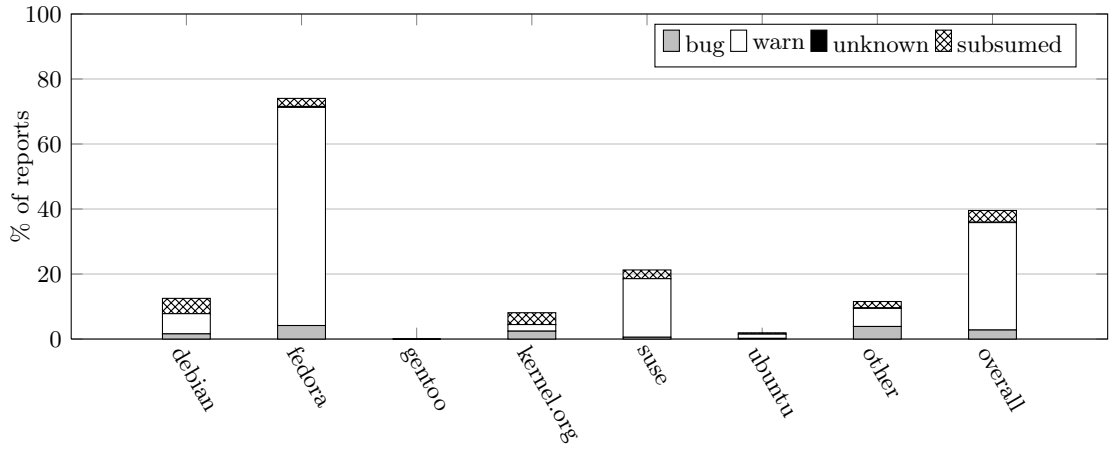


Figure 6: Duplicate reports per Linux distribution

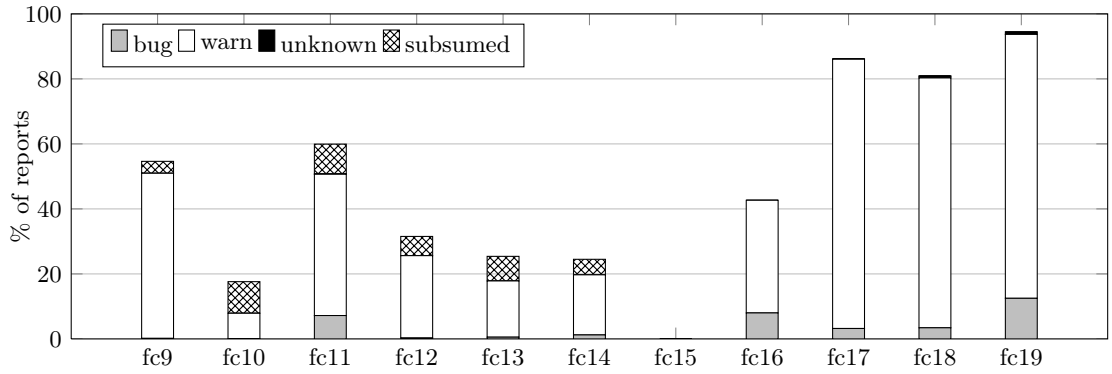


Figure 7: Duplicate reports per Fedora release

timestamps before computing the SHA1 hash. *iii.* Finally, a prefix or suffix of a report may be missing. For duplicate detection, we parse the reports, and consider to be duplicates reports for which the only difference as compared to another oops is that some fields are missing. We refer to such reports as being *subsumed*.

Fig. 6 shows the percentage of reports that may be duplicates of another oops for each Linux distribution, considering those found to be identical by the SHA1 hash and by subsumption. Overall, we observe a rate of duplicates of 40%. Furthermore, we observe that although Fedora has used ABRT, having a sophisticated duplicate detection mechanism, since Fedora 12, many of the Fedora reports, particularly the warnings, appear to be duplicates. Recall, however, that ABRT discards the oops id, so we are missing a crucial piece of information that could distinguish reports. Fig. 8 shows the percentage of oopses with no oops id, for versions having at least 1000 reports. Finally, Debian has the largest number of duplicates, at over 10,000. However, it has the largest number of reports overall (Fig. 5), and so this number makes just over 12% of its total.

In summary, Linux distributions that are supposed to result in few duplicates, seem to result in many, although the oops id is missing so we cannot be sure. And Linux distributions that are supposed to be likely to result in duplicates do result in duplicates, but at a relatively low rate. Thus, subsequently, we consider all reports, whether or not they appear to be duplicates. This

strategy may magnify the importance of some kinds of oopses, in particular warnings, which are the most common possibly duplicated reports, but it does accurately reflect the kind of information that is provided to the user of the repository.

Missing oopses It is intrinsically difficult to accurately count the number of missing reports, because such reports are not present in the repository. Nevertheless, we can estimate their number from the various counters and historical information that are found in some kinds of reports (see Table 1). Specifically, we consider *i.* the oops id, *ii.* the die counter, and *iii.* the taint. The oops id is a random number generated on the first oops id generating oops on each boot and incremented for each subsequent such oops. Because the initial value is random, gaps may indicate either missing reports or the starting value for a new boot. Thus, the oops id itself is not a reliable means of detecting missing reports. The die counter is initially 0, and is incremented and then printed on several kinds of oopses. When the die counter is greater than 1, there should be a preceding oops in the same boot with the preceding die counter value. Any such oops id that is not present represents a missing oops. Finally, the taint word in an oops indicates whether there is at least one previous bug or warning within the current boot. For a report with either 'D' (die) or 'W' (warn) taint and with an oops id, the immediately previous oops id should also be present. Again, any such oops id that is not present represent a missing report.

Over all of the versions, we find 7852 oopses where there is an oops id and the die counter reaches a value greater than 1, and 51% of these where the expected previous die counter is missing. Furthermore, we find 37265 oopses where there is an oops id and the taint indicates a previous bug or warning, and 47% of these where the expected previous oops id is missing. In each case, we require that the version information be present and identical, to try to avoid confusion between oops ids that are identical but unrelated. These results consistently show a high rate of missing oopses in cases where multiple oopses occur within a single boot.

While our results suggest a high rate of missing oopses, the validity of this conclusion is called into question by the small number of oopses taken into account: 4% in the die counter case and 20% in the taint case. The scope of our analysis is indeed constrained by the need to take into account the oops id. As we have noted, Fedora's ABRT, used starting with Fedora 12, removes the oops id from a kernel oops. While only 0.2% of the reports have a die counter greater than 1 but no oops id, 11% have taint indicating a previous bug or warning but no oops id. Thus, it is impossible to assess missing oopses in these cases. Nevertheless, based on the ratio of the number of oopses missing according to the taint information to the total number of oopses plus the missing number, it appears that overall, we are missing at least 9% of the oopses.

The oops id In principle, the problem of detecting duplicate and missing reports could be resolved if the kernel would associate a unique identifier with each boot, and a counter, analogous to the die counter, that is incremented on each oops. A step in this direction is already present with the oops id. Nevertheless, some observations about the oops id reveal challenges that can be encountered in implementing such a scheme.

To be useful, the proposed boot identifier must be unique. We have observed that the current oops id is not always unique. In the extreme, the oops ids 4eaa2a86a8e2da22, a7919e7f17c0a727, and 0000000000000002 occur with 35, 27, and 21 different version strings, respectively. Overall, we have 28 such non-unique oops ids. Some may be due to corruption of the oops id counter. Others may be due to incrementing such a corrupted value; for instance, we also find 0000000000000003, etc., up to 0000000000000008, in decreasing amounts. Still it is possible that some of these duplicated oops ids result from weaknesses in the random number generator.

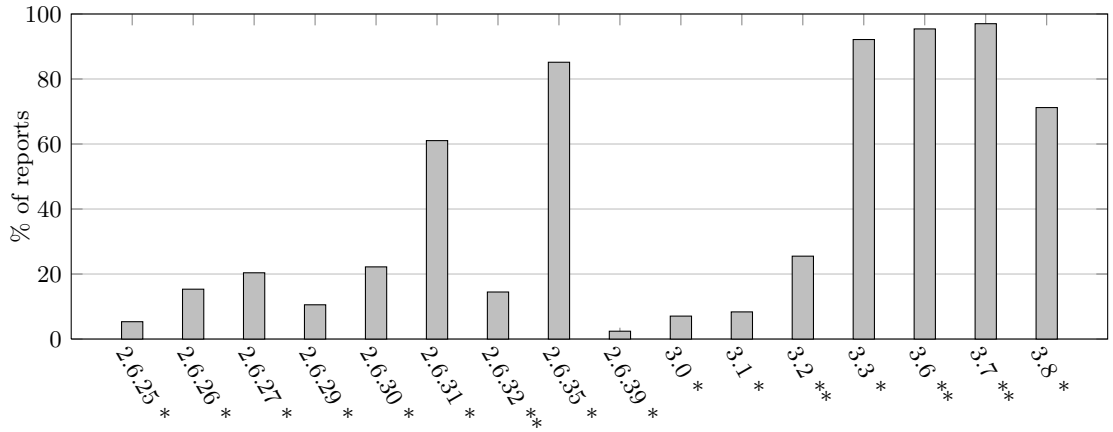


Figure 8: Percentage of reports without oops ids per Linux version. Version with one star have at least 1000 reports, while versions with two stars have at least 10,000 reports. This convention is followed in subsequent figures.

Starting in Linux 3.3,[2] the implementation of the Linux random number generator was improved to take advantage of hardware-level random number generation facilities, if available. Such facilities are available for the x86 architecture. Indeed, after Linux 3.3, our only duplicates, taking into account the version string, are 0000000000000002 and nearby oops ids, as well as a7919e7f17c0a727. These may thus represent corrupted oops id values. On the other hand, as shown in Fig. 8, we also have relatively few oopses for those versions that contain oops ids.

Likewise, for the proposed counter to be useful, the process of incrementing it on each oops must be well-defined. Currently, in the Linux kernel, neither the oops id nor the die counter is incremented atomically, meaning that these computations are subject to race conditions. Indeed, introducing locks could reduce the amount of information available, *e.g.*, if the lock itself were to become corrupted due to the error.

Finally, in the context of the repository, the boot id and counter have to be transmitted to the repository reliably. This is not he case of recent Fedora oopses, where ABRT strips the oops id.

4 Correlation with external information

To help establish the representativeness of the data in the kernel oops repository, we compare the properties of the oops data to some independent observations about the Linux kernel. In this, we study the kernel versions and Linux distributions associated with the oops reports, and the architectures from which the reports originate.

Linux version As shown in Fig. 4, we have at least 1000 reports from Linux kernel versions ranging from 2.6.25, released in April 2008, to 3.8, released in February 2013. Four versions, from Linux 3.6 to Linux 3.9, were released in the period covered by our study. For the recent versions, we might expect that the number of reports would increase when the version is released, then remain stable for the period in which it is the most recent version, and then finally decrease when a new release appears. Earlier versions are typically used by users who have a need for stability,

[2]Kernel patch cf833d0b9937874b50ef2867c4e8badfd64948ce

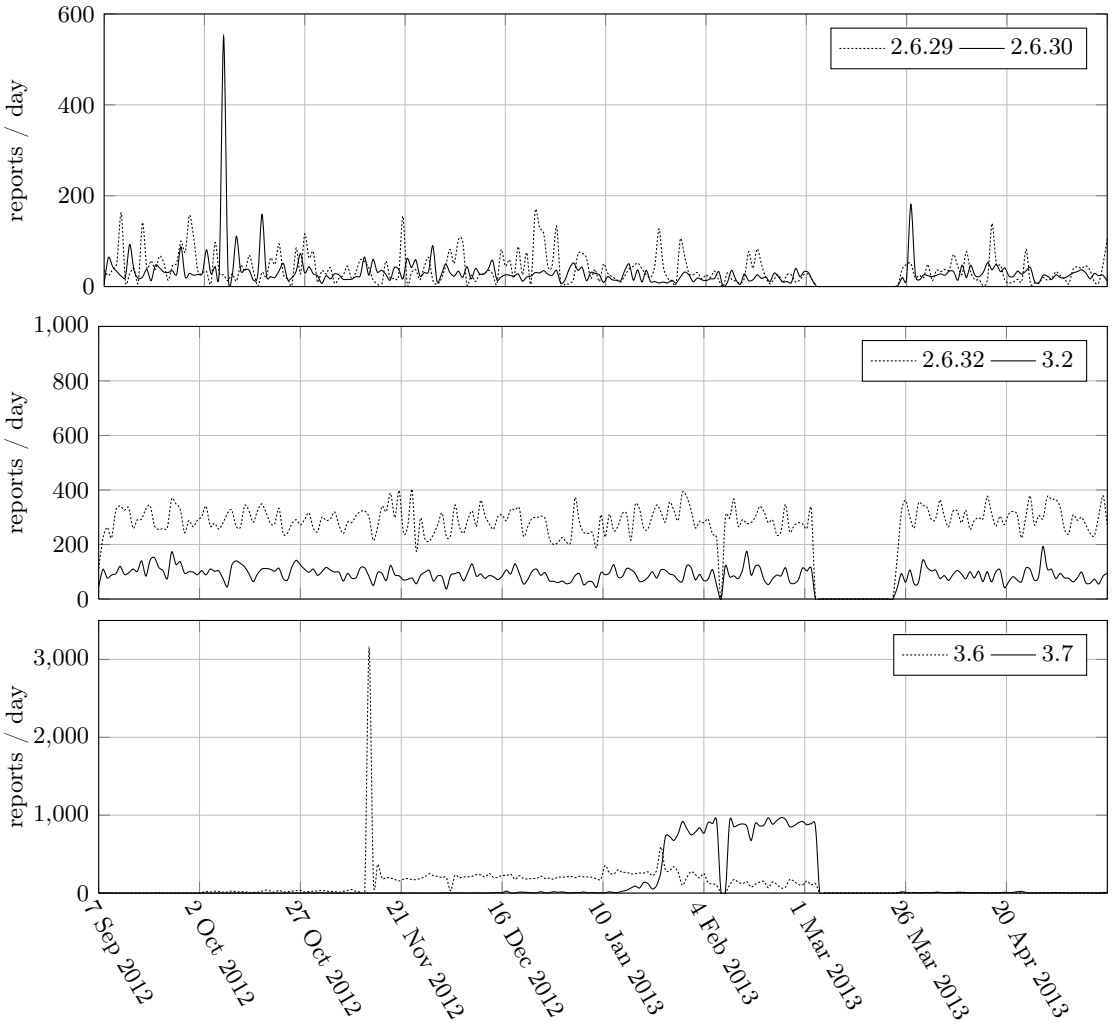


Figure 9: Prevalence of reports from selected Linux versions over time (versions for which there are at least 5000 reports)

rather than new functionality. We might expect the number of reports from such versions to be fairly stable, but to decrease over a long period of time.

The top two graphs of Fig. 9 show the number of reports per day for the older versions for which we have the most reports: 2.6.29, 2.6.30, 2.6.32, and 3.2. For these versions, the number of reports per day is fairly stable, modulo a few spikes and periods in which there are no reports at all. Excluding the days on which we have no reports, 2.6.29 has on average 39 reports per day, 2.6.30 has on average 32 reports per day, and 2.6.32 has on average 287 reports per day. In each case the number of reports per day is relatively constant across the considered time period. These results substantiate our hypothesis. However, for Linux 3.2, we have only 90 reports per day, which is fewer than for Linux 2.6.32, even though Linux 3.2 is more recent. Thus, it appears to be necessary to take into account other factors than just the age of the Linux version.

The bottom graph of Fig. 9 shows the number of reports per day for the recent versions for which we have the most reports: 3.6 and 3.7. As expected, the number of reports for Linux 3.6

increases sharply in early November 2012, shortly after the release of Linux 3.6 at the end of September 2012, and the number of Linux 3.6 reports starts to decline in late January 2013, as the number of reports from Linux 3.7, which was released in December 2012, rises. Note that in each case, reports start to appear in large numbers roughly one month following the release date. This can be ascribed to the fact that users typically do not adopt each version as it is release, but wait for the subsequent more stable subreleases, in which essential bug fixes have been applied.

While versions 3.6 and 3.7 fit the expected pattern, we do not have the same pattern with Linux 3.8, which was released in mid February 2013, well within the considered period. We have very few reports for this version, typically 25 or fewer per day. Again, we conclude that other factors must be involved.

Linux distribution To better understand the reason for the number of reports per version, we consider also the association of the various Linux versions with the various Linux distributions (*e.g.,* Fedora, Debian, Ubuntu etc.). Indeed, most users of the Linux kernel use the kernel provided with their Linux distribution, and upgrade the kernel as the distribution suggests to do so.

Fig. 10 shows the prevalence of kernel oopses for each Linux kernel and for each Linux distribution, for those versions for which we have at least 1000 reports. Most versions have most reports from either Debian or Fedora, but not both. Reports from Linux 2.6.26, 2.6.32, and 3.2 are almost entirely from Debian. Linux 2.6.26 was the kernel of Debian 5.0, released in 2009, Linux 2.6.32 was the kernel of Debian 6.0, released in Debian 6.0, released in 2011, and Linux 3.2 is the kernel of Debian 7.0, released in 2013. In particular, within the considered time period, Debian 6.0 was the current "stable" version, used by most Debian users, and Debian 7.0 was the current "testing" version. This explains the constant, relatively high number of reports for Linux 2.6.32, observed in Fig. 9, and the constant but lower number of reports for Linux 3.2. We expect that the number of reports for Linux 3.2 will rise, now that it is used in the current Debian "stable" version, as of May 2013.

As shown in Fig. 10, many of the versions not used by the Debian releases mostly have reports from Fedora. Fig. 11 breaks down all of the versions for which we have any report from Fedora by the rate of the various Fedora releases as compared to the number of Fedora reports for that version. We find that the earliest Linux version associated with each Fedora release in the oopses is always the one that was shipped with that Fedora release. For instance, oopses from Fedora 9 appear with Linux 2.6.25, which is the default kernel of Fedora 9. Furthermore, we also see how the Linux kernel shipped with a Fedora release changes over time, and then how one Fedora release is subsumed by the next one. Fedora 9 oopses occur with Linux 2.6.25 through 2.6.27, at which point they are replaced by oopses from Fedora 10. 2.6.27 is indeed the default kernel of Fedora 10. Thus, the oopses associated with Fedora releases correspond to the versions associated with those releases.

Hardware architecture Fig. 12 shows the number of bug and warning reports found in the repository for the 32-bit and 64-bit x86 architectures. We determine the architecture by the size of instruction addresses in the call trace and the values in the registers, whichever is available. We furthermore only consider reports for which which a version and error type are available, representing 91% of the total number of reports. There are few reports for versions 2.6.18 and 3.9, and thus the information for these versions may not be representative.

We expect that the number of users of 32-bit architectures is declining, and thus there should be fewer reports from such architectures for more recent kernels. Indeed, starting with version 2.6.36, the rate of reports from 32-bit machines is almost always much lower than the rate of

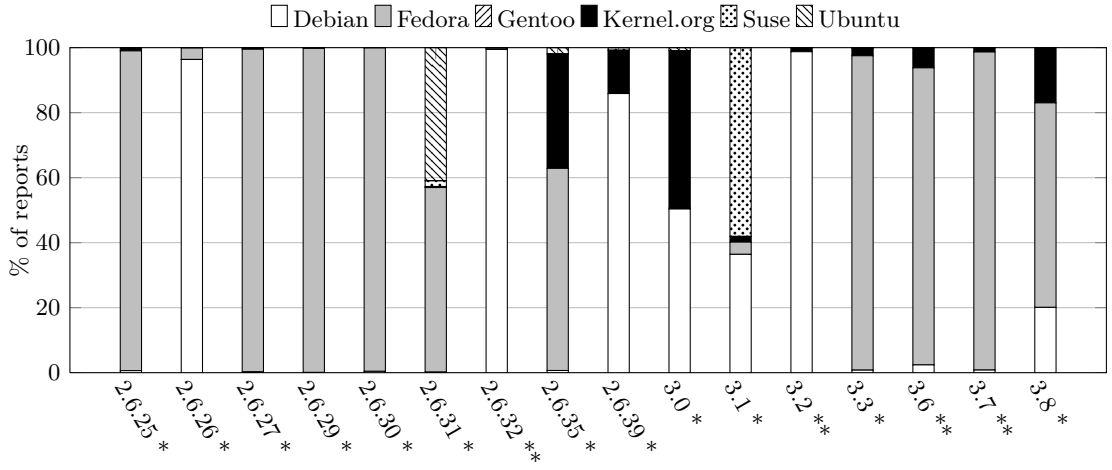


Figure 10: Prevalence of reports from different distributions for different versions, for which there are at least 1000 reports (* referred to Fig 8)

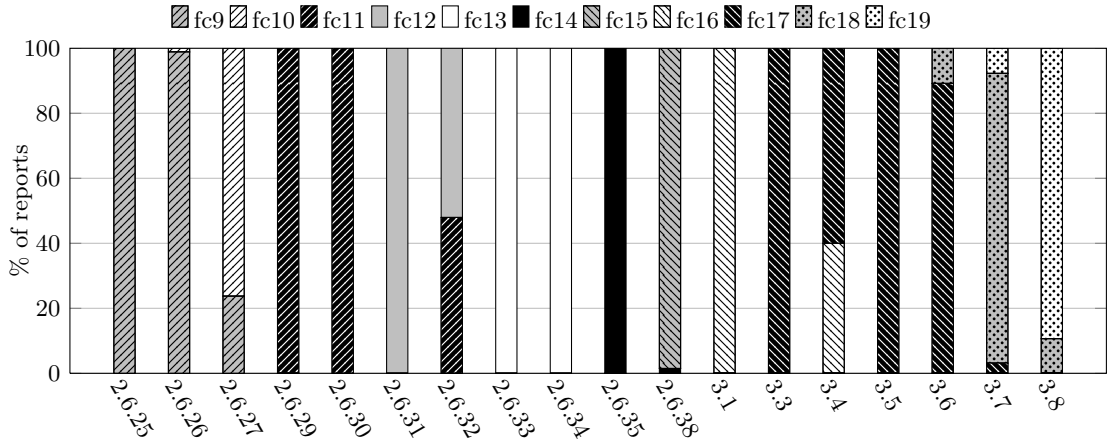


Figure 11: Prevalence of reports from different Fedora distributions for different versions (all versions for which there is at least one Fedora report are included)

reports from 32-bit machines from versions prior to 2.6.36, particularly in terms of the rate of reports that represent bugs. Most Linux x86 code is shared between both the 32 and 64-bit architectures, and so this drop off is not likely to be due to an improvement in the quality of the 32-bit specific code. Thus, we conclude that the recent releases of Linux have fewer 32-bit reports simply because the 32-bit architecture has fewer users.

5 Features related to kernel reliability

We now consider how to correctly interpret various features of an oops that may be useful in assessing kernel reliability.

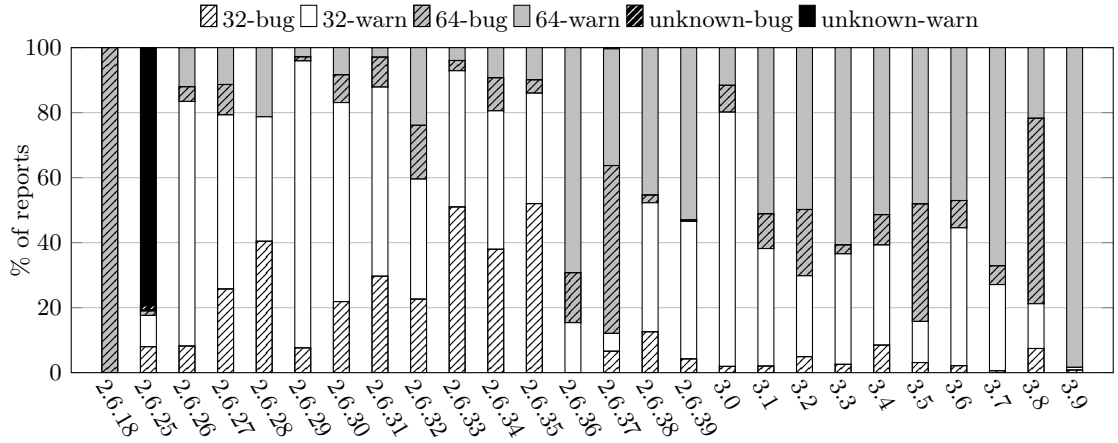


Figure 12: Prevalence of reports from 32 and 64 bit machines

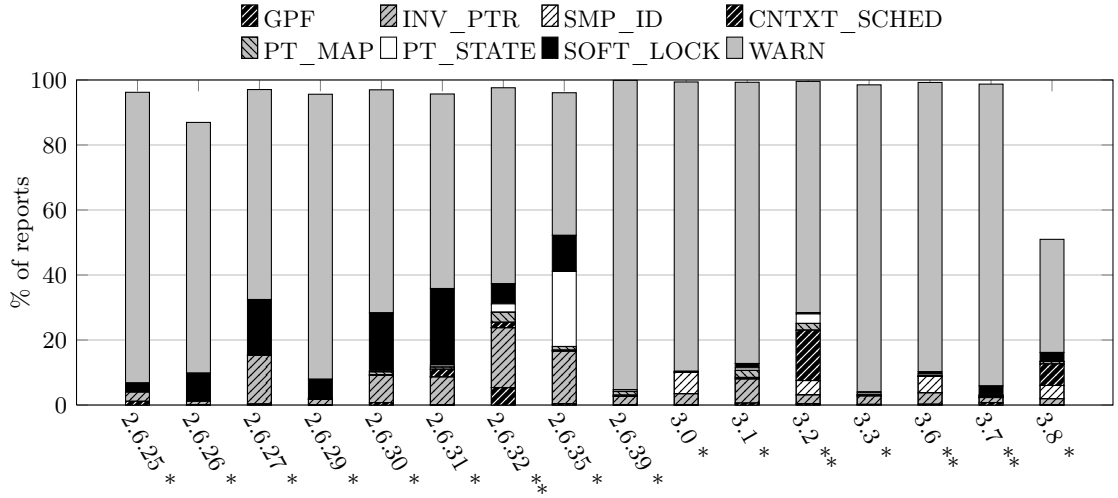


Figure 13: Prevalence of the 8 most common events (bugs or warnings)

Error types Fig. 13 shows the frequency of various bugs and warnings, across the various versions, for the oopses specifying a version. In every case, warnings make up the largest share of the oopses. Many of these warnings come from the same functions. Fig. 14 and 15 show that in some versions, a single function may be the source of over 90% of the warnings, for either the 32-bit or 64-bit architecture. We furthermore observe that it is useful to distinguish between these architectures, because some kinds of warnings that are prominent on one architecture are less common or absent on others. For example, the warning in `default_send_IPI_mask_logical` is common on the 32-bit architecture, but the function is not used on the 64-bit architecture.

Call trace top origin Several works on Linux code have identified drivers as a main source of faults, *e.g.*, based on the results of static analysis [7, 8]. A kernel oops repository provides an opportunity to determine whether drivers are the main source of errors encountered in practice. We consider invalid pointer references, a common kind of serious error (Fig. 13), and study the

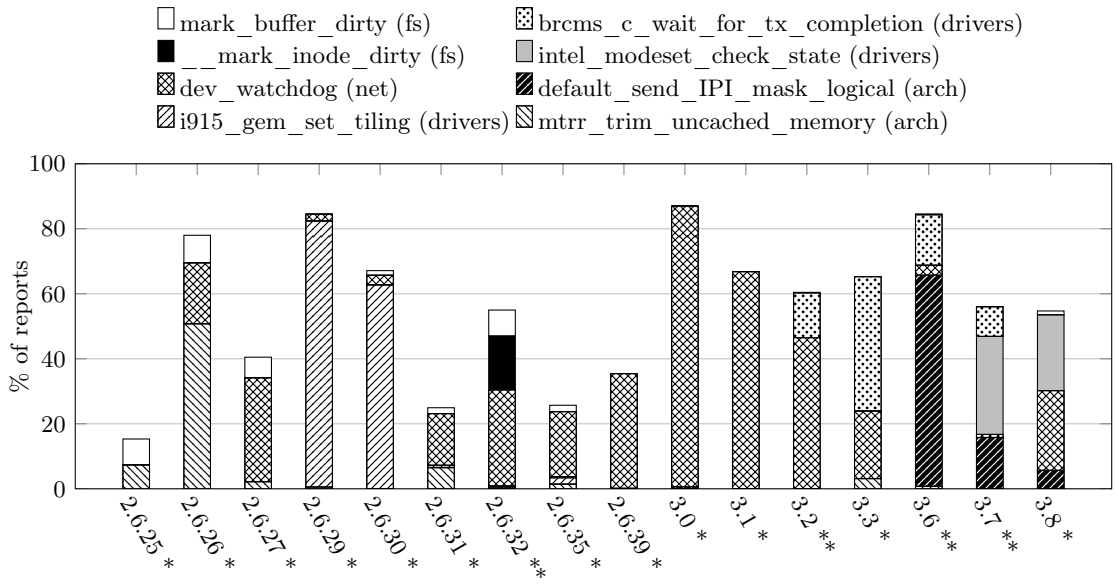


Figure 14: Top 8 warning-generating functions, 32-bit architecture

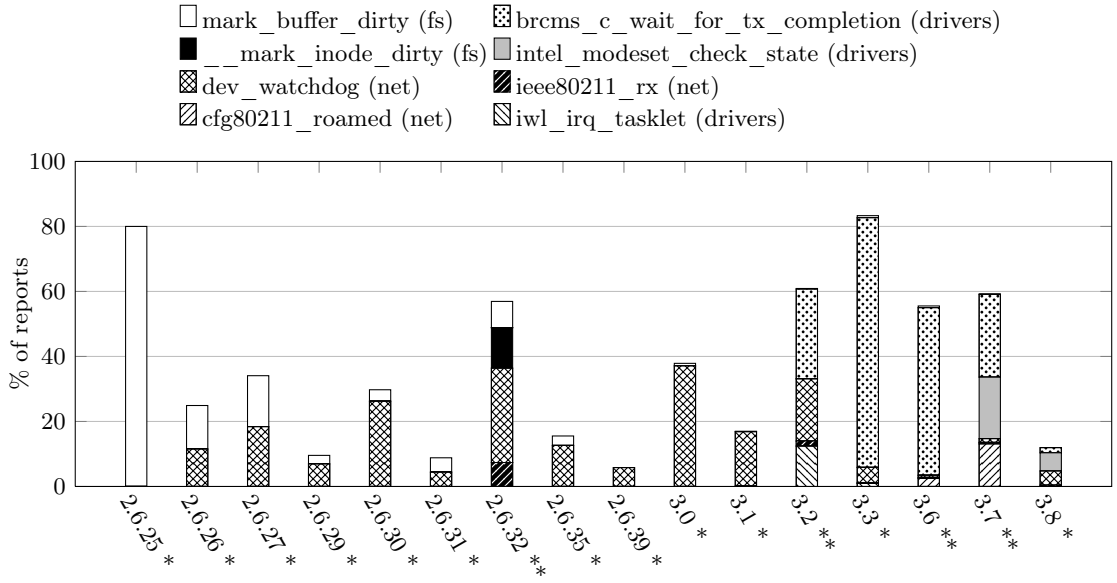


Figure 15: Top 8 warning-generating functions, 64-bit architecture

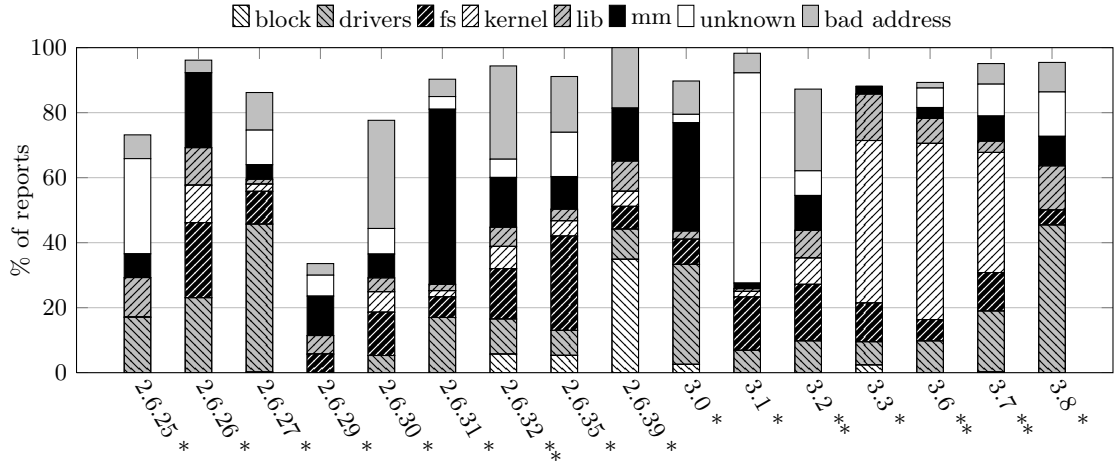


Figure 16: Top 6 services containing invalid pointer references

service associated with each oops. We approximate the service as the top-level subdirectory containing the definition of the function in which the invalid reference occurs.

Fig. 16 shows the top 6 services in which invalid pointer references occur. Additionally, there are a number of cases where the service is unknown because the invalid reference occurs in an external function, that is not part of the kernel source tree ("unknown"), and where the service is unknown because the call to the function itself represents the invalid reference ("bad address").

While driver functions do make up a large percentage of the functions in performing an invalid pointer reference, such errors occur in other services as well. Notably, in Linux 3.3, 3.6, and 3.7 there are many errors in `kernel` code. To understand why, we have also studied the names of the functions in which invalid pointer references most frequently occur. One such function is the primitive locking function `_raw_spin_lock`, which is defined in the `kernel` subdirectory. Because locking is such a basic functionality of the Linux kernel, it is not likely that the fault is in the definition of `_raw_spin_lock` itself, but rather in its caller, which may have attempted to lock an invalid lock. To identify the service associated with the caller, we search for the name of the calling function at the top of the call trace. We find that 74% of the failing calls to `_raw_spin_lock` come from file system functions.

Correctly using the call trace to find the caller of the crashing function, however, requires understanding the kernel call trace generation process. As an optimization, the kernel is compiled such that the stack frames do not contain a frame pointer. This optimization implies that the kernel is not able to unambiguously identify return pointers on the stack. Instead, it scans the stack to find addresses that could correspond to an address within a function. For each such address, it determines whether its position corresponds to the expected offset from its stack base pointer register. If this property is not satisfied, the pointer is referred to as *stale* and the name of the corresponding function is annotated with `?` in the *call trace* field of kernel oops. *Stale pointers* are common, and may arise when function pointer arguments are passed via the stack or when the stack contains uninitialized local variables whose locations coincide with the positions of previously stored return pointers. It may also occur that the base pointer register does not correspond to any of the stack positions containing return pointers, and thus the entire call trace is considered to be stale. The latter phenomenon is illustrated by the call trace in our sample kernel oops in Fig. 1 (lines 24 to 31).

To address the issue of stale pointers, when consulting the call trace, we find the service

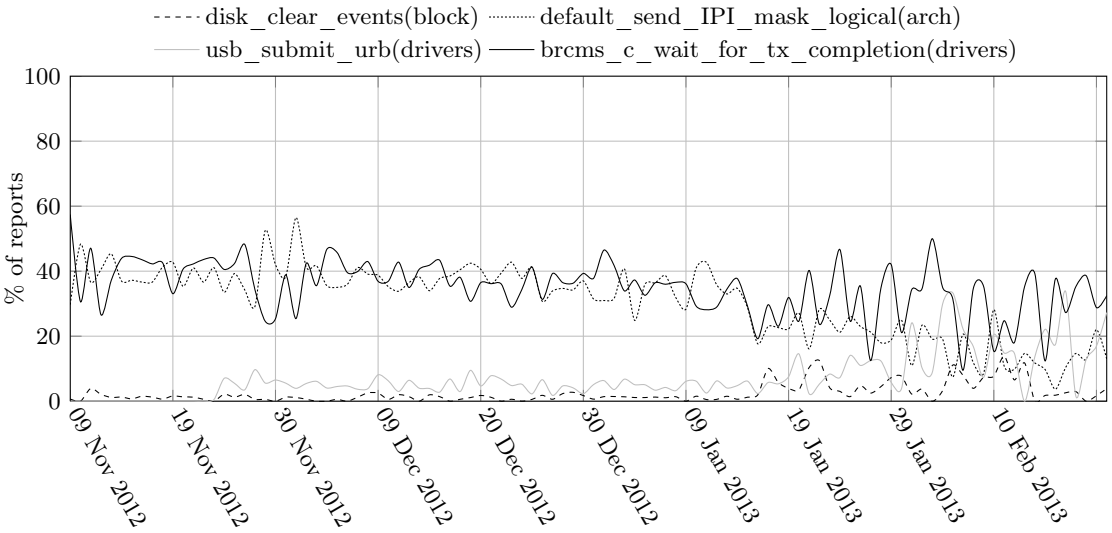


Figure 17: Common warnings by day for Linux 3.6

associated with the topmost non-stale pointer, if available. If there is no non-stale pointer, we fall back on taking the function at the top of the call trace.

Spikes As shown in Fig. 9, there are two significant spikes in the oopses per day for Linux 3.6, on November 9, 2012 and January 16, 2013. Such spikes can potentially dominate other data and distort assessments of reliability. As shown in Fig. 13, most of the reports for these versions are warnings. Thus, we study the frequency of warnings by day for Linux 3.6 to see if the number of oopses per day has an impact on the frequencies of these warnings.

Fig. 17 shows the rate of warnings from the most common warning-generating functions during the period in which there are the most oopses for Linux 3.6. While the rate of warnings from the most common warning-generating functions changes over time, there is no significant difference between November 9 or January 16 and the surrounding days. Thus, the spikes on these days affect the number of reports present, but not their relative number, and the information for these days can be safely combined with the rest.

Trigger action We next consider the action that caused the kernel to be entered, which provides another perspective on which kernel services are error prone. Actions include those that are initiated by the kernel, such as booting and creating a kernel thread, those that are initiated by applications, such as system calls, and those that are initiated by devices, such as interrupts. To identify the trigger action, we analyze the bottom of the call trace. In doing so, we must take into account stale pointers, as was described above. But we also must take into account the particular structure of Linux call traces.

For both 32-bit and 64-bit x86 architectures, Linux manages a linked list of stacks, comprising a process stack, an interrupt stack, and, on 64-bit x86 architectures, a set of exception stacks.[3] For the 32-bit x86 architecture, there is an explicit annotation of an interrupt stack only if both an interrupt stack and a process stack are present. Thus, there is an ambiguity; if an interrupt occurs when no process is running in the kernel, then there is only one stack and no delimiter.

[3]On the 64-bit x86 architecture, a serious interrupt is referred to as an exception.

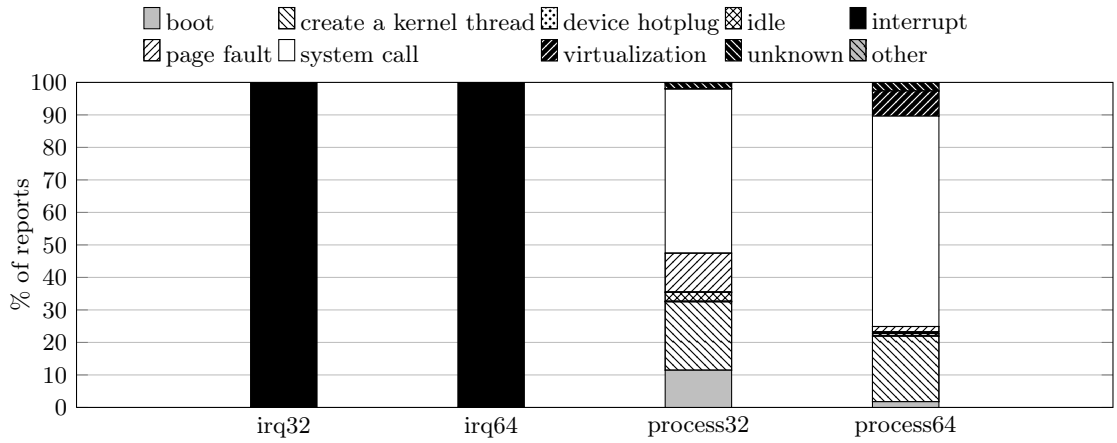


Figure 18: Stack bottoms of interrupt and process stacks

We thus classify a 32-bit call trace containing only a single stack as representing an interrupt stack if the bottom function of the call trace is an interrupt function, such as `common_interrupt` or `apic_timer_interrupt`. For the 64-bit x86 architecture, interrupt stacks are delimited by `IRQ` and `EOI` (End Of Interrupt) and exception stacks are delimited by the name of the exception and `EOE` (End Of Exception). There is no ambiguity in this case.

To identify the action that triggered an error, we consider the bottom-most non-stale function in the current stack, if any, or if there is no non-stale pointer, then the bottom-most stale function in the current stack. This strategy covers 89% of the reports, excluding those for which there is no call trace or for which the architecture cannot be determined. We have manually classified these functions according to what kind of action they represent: system boot, creation of a kernel thread, the idle process, an interrupt, a page fault, a system call, or support for virtualization. Fig. 18 shows the results for interrupt and process stacks for the 32-bit and 64-bit x86 architectures, considering only those functions that occur at least 50 times. We have only 420 occurrences of an exception stack, and none of the stack bottoms satisfied this threshold.

Most of the results are as could be expected: the trigger action for an interrupt stack is always an interrupt, and the most common trigger action for a process stack is a system call. We do have a small anomaly, that is not visible in the graph, for 64-bit process stacks. Although the interrupt stack is in principle unambiguous for the 64-bit x86 architecture, we have over 350 cases where an interrupt function appears at the bottom of what seems to be a 64-bit x86 process stack. All of these instances come from Suse. Indeed, only 5 Suse reports out of the 1139 having a call trace have an interrupt stack delimiter, and thus we conjecture that the Suse oops reporting tool may eliminate them.

Impact of taint Because the Linux kernel is intended to be a long-running system and because it must meet stringent performance requirements, it cannot keep available in memory an unbounded history of its actions. Thus, an oops primarily provides an instantaneous picture of the state of the kernel. Nevertheless, when certain kinds of events have occurred in the kernel execution, the kernel may have been left in an unreliable state. Thus, a small amount of historical information is recorded, known as *taint*, which amounts to a one-word bit map recording the previous occurrence of various events. The events considered involve the inclusion of suspect code in the kernel, and the previous occurrence of specific and generic errors. The presence of

Table 2: Taint types

Loaded code:	**Specific errors:**
P: Proprietary module loaded.	M: Machine check exception.
G: No proprietary module loaded.	B: System has hit bad_page.
F: Module forcibly loaded.	U: Userspace-defined naughtiness.
R: Module forcibly unloaded.	A: ACPI table overridden.
C: drivers/staging modules loaded.	I: Severe firmware bug.
O: Out-of-tree modules loaded.	S: SMP with non-SMP CPU.
Generic errors:	
D: Previous die.	W: Previous warning.

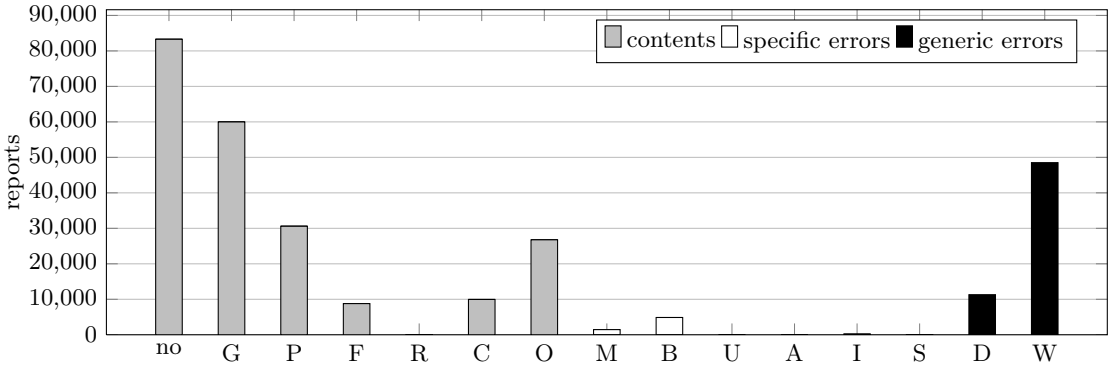


Figure 19: Number of occurrences of individual taint bits

taint can reduce the credibility of an oops as a record of an error in the kernel.

Fig. 19 shows the number of occurrences of the individual taint bits across the reports. Only reports that have taint information are included, amounting to 93% of the reports. Almost half of these reports have no taint (label 'no'). For these oopses, there is the greatest likelihood that the source of the problem is captured in the oops, rather than depending on some previous action. Of the remainder, two thirds ('G') do not involve proprietary modules, meaning that maintainers can reconstruct the execution environment using code that is freely available. Of those that do not involve proprietary modules, 45% do involve external modules ('O'), implying some extra work for the maintainer to find the relevant code. Finally, 6% of the reports containing taint information derive from kernels that include so-called "staging" drivers ('C'). These immature drivers are integrated into the kernel so that they can keep up with kernel changes as they mature. The kernels involved may be development kernels. Few of the reports come from kernels that have previously experienced the specific errors represented by the taint field ('M' through 'S'); only the numbers of machine check exceptions ('M') and bad pages ('B') are non-negligible. Finally, 6% of the reports containing taint information come from kernels that have previously experienced a serious error ('D'), and 28% come from kernels that have previously experienced a warning ('W').

6 Threats to Validity

The overall goal of our study is to identify the potential threats to validity in using kernel oopses as an indicator of Linux kernel reliability. Our study itself, however, may suffer from threats to validity, related to our interpretation of the studied data. Several of the authors have experience developing and maintaining Linux kernel code [9, 10, 11, 12], we have studied the source code of the user-level oops-reporting tools, and we have duplicated the process of generation and

submission of kernel oopses to the repository. This experience helps ensure the validity of the analysis.

7 Related Work

Windows Error Reporting (WER) [1, 3] is a post-mortem debugging system that collects and processes error reports from a billion machines. The primary goal of WER is to help programmers prioritize and debug the issues reported directly from the end users, which coincides with a potential use of Linux kernel oopses. WER enables *statistics-based* debugging, which helps prioritize debugging effort, find hidden causes, test root cause hypotheses, measure deployment of solutions, and watch for regressions. Our work addresses how to accurately interpret a repository of error reports for the Linux kernel. This analysis opens the door for using Linux kernel oopses in the context of such a debugging system.

Bug reports are another form of artifact that reflect the bugs encountered by real users of a software system. Jalbert *et al.* [13] studied the rate of duplicate bug reports within a dataset of 29,000 bug reports from the Mozilla project. Bug reports, however, are different than oopses, in that bug reports are written by users in free text, making detecting duplicates much more difficult. The kinds of duplicates are furthermore different in nature than the ones that we have considered in Section 3. Jalbert *et al.*'s goal is to detect cases where multiple users encounter the same problem, while we are interested in detecting cases where a tool has submitted a report generated by a single error occurrence multiple times.

Seo and Kim use Mozilla crash dumps to study the problem of incomplete bug fixes [14]. The Mozilla Crash Reporting System groups crash reports based on the name of the crashing function. Seo and Kim analyze a bug fix proposed for a given crashing point to determine whether the fix covers all of the collected stack traces that reach that point. A similar study could be carried out for Linux if the oops call traces can be interpreted correctly. We have highlighted the issues of stale pointers and multiple stacks (Section 5) that must be taken into account when interpreting a call trace.

Potential run-time errors in Linux have been studied using fault injection. Gu *et al.* [15] performed a series of fault injection experiments on the Linux kernel of version 2.4.20. Their goal was to analyze and quantify the response of the Linux kernel in various crashing scenarios. Most recently, Yoshimura *et al.* [12] did an empirical study on the kernel oopses resulting from fault injection. Their goal was to quantify the damage that could be incurred at the time of a kernel oops. These illustrate the potential for using kernel oopses in reliability studies. Our work opens the door for using kernel oopses generated by real users, rather than oopses generated by fault injection.

A number of studies have used static analysis to estimate the rate of faults within the source code of the Linux kernel [7, 8]. The latter study has found that the quality of the Linux kernel code has improved significantly, *e.g.,* the number of faults per line has dropped, but still, the Linux kernel is far from bug-free. Thus, the study of Linux kernel oopses will continue to be relevant.

8 Conclusion

In this paper, we have provided a tour of the Linux kernel oopses available in the recently established oops repository at Red Hat. Our analysis has revealed the information available in such a repository, but also the challenges involved in interpreting it. As studies of repositories such as those of Windows and Mozilla have shown, an oops repository has great potential to

shed light on the reliability of the Linux kernel, as well as to help study development practices. We expect that our results will serve as a guideline for the correct interpretation of the results of future research studies.

Acknowledgement

We would like to thank Anton Arapov for granding us the access to the kernel oops repository. The experiments carried out in this paper relied heavily on the OCaml parallelization library Parmap, developed by Danelutto and Di Cosmo [16].

References

[1] Microsoft MSDN. Windows Error Reporting.

[2] Apple Inc., CrashReporter. Technical Report TN2123, Cupertino, CA, 2008.

[3] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt, "Debugging in the (very) large: ten years of implementation and experience," in *SOSP*, Big Sky, Montana, USA, 2009, pp. 103–116.

[4] Automatic Bug Reporting Tool (ABRT), Fedora.

[5] Kernel Oops Tracker (kerneloops), Ubuntu.

[6] A. Arapov. (2012, Sep.) Kernel Oops Repository.

[7] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *SOSP*, Banff, Alberta, Canada, 2001, pp. 73–88.

[8] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller, "Faults in Linux: ten years later," in *ASPLOS*, Newport Beach, CA, USA, Mar. 2011, pp. 305–318.

[9] G. Muller, J. L. Lawall, and H. Duchesne, "A framework for simplifying the development of kernel schedulers: Design and performance evaluation," in *HASE 2005 - High Assurance Systems Engineering Conference*, Heidelberg, Germany, Oct. 2005, pp. 56–65.

[10] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in Linux device drivers," in *EuroSys*, Glasgow, Scotland UK, Apr. 2008, pp. 247–260.

[11] S. Saha, J.-P. Lozi, G. Thomas, J. L. Lawall, and G. Muller, "Hector: Detecting resource-release omission faults in error-handling code for systems software," in *DSN13*, Glasgow, Scotland UK, Jun. 2013.

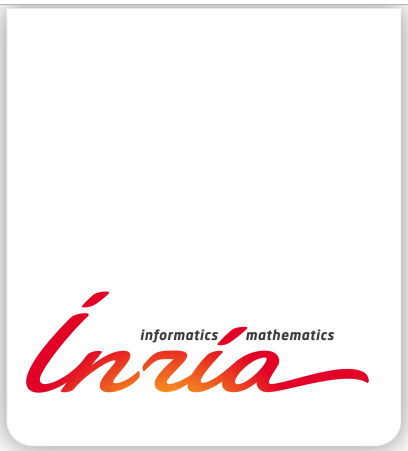
[12] T. Yoshimura, H. Yamada, and K. Kono, "Is Linux kernel oops useful or not," in *HotDep*. Hollywood, CA, USA: USENIX Association, Oct. 2012.

[13] N. Jalbert and W. Weimer, "Automated duplicate detection for bug tracking systems," in *DSN*, Jun. 2008, pp. 52–61.

[14] H. Seo and S. Kim, "Predicting recurring crash stacks," in *ASE*, Essen, Germany, 2012, pp. 180–189.

[15] W. Gu, Z. Kalbarczyk, K. Ravishankar, and Z. Yang, “Characterization of Linux kernel behavior under errors,” in *DSN*, San Francisco, CA, USA, Jun. 2003, pp. 459–468.

[16] M. Danelutto and R. Di Cosmo, “A "minimal disruption" skeleton experiment: Seamless map & reduce embedding in ocaml,” *Procedia CS*, vol. 9, pp. 1837–1846, 2012.



RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399